基于程序频谱的两阶段缺陷定位方法 *

伍 佳，洪 玫†，万 莹，邓惠心，潘春霞

(四川大学 计算机学院，成都 610065)

摘 要：缺陷定位是软件质量保证中关键且困难的一项工作。随着软件规模增大，人工进行缺陷定位的成本越来越高，自动化的缺陷定位技术成为研究热点。现有的基于程序频谱的缺陷定位技术可以将缺陷定位到程序语句，但对于大型复杂的软件系统，这种定位方法将带来较大的时间花销。针对此问题，提出一种基于程序频谱的两阶段缺陷定位方法，第一阶段为粗粒度定位，将缺陷定位到程序模块；第二阶段为细粒度定位，在定位的程序模块中，再将缺陷定位到语句；最后输出可疑语句推荐列表，辅助开发人员的调试工作。实验结果表明，相比于传统的方法，该方案在保证定位效果的前提下，平均减少了 10.24%的定位时间。

关键词：缺陷自动定位；程序频谱；测试用例；软件调试

中图分类号：TP311 doi: 10.19734/j.issn.1001-3695.2020.03.0065

Two-phase fault localization method based on program spectrum

Wu Jia, Hong Mei†, Wan Ying, Deng Huixin, Pan Chunxia

(*School of Computer Science*, *Sichuan University*, *Chengdu Sichuan* 610065, *China*)

Abstract: Fault localization is a critical and difficult task for software quality assurance. With the increasing of software scale, the cost of manual fault localization becomes higher and higher, so that many research tasks focus on automatic fault localization. Existing methods of program spectrum based fault localization locate fault to program statements, they can be time consuming in large and complex software. To solve this problem, this paper proposed a two-phase fault localization method based on program spectrum. The first phase was coarse grained localization, located fault to program modules. The second phase was fine grained localization, located fault to program statements in the suspicious modules. Finally, the recommendation list of suspicious statements was obtained to assist the developer in debugging. Experimental results show that comparing with traditional methods, this method can guarantee the effect of localization and reduce localization cost by 10.24% on average.

Key words: automatic fault localization; program spectrum; test case; software debugging

0 引言

在软件质量保证中，传统的缺陷定位要求开发人员多次设置程序断点，执行失败的测试用例，观察程序变量取值，直至找到缺陷根原因。自动化缺陷定位方法依据缺陷程序相关的数据，比如源代码、测试套件、测试记录、缺陷报告等，采用数据分析技术，发现可疑的缺陷位置，辅助开发人员找到缺陷根原因。相比于传统方法，现代技术借助于计算机的处理能力，进行缺陷数据分析，实现了缺陷定位的自动化，减少了定位成本，提高了定位效率。

在自动化缺陷定位方法中，根据是否需要执行测试用例，可分为静态缺陷定位[1~4]、动态缺陷定位[5~16]以及多种方法相结合的缺陷定位[17,18]。静态缺陷定位常使用信息检索[1,2]、机器学习[3,4]等技术分析日志、缺陷报告等信息进行定位。动态缺陷定位通过执行测试用例获取相关信息，使用程序切片[5,6]或程序频谱[7~16]等技术来确定可能含有缺陷的程序片段。多种方法相结合的缺陷定位通常组合以上多种定位技术来提高缺陷定位的准确性。在动态缺陷定位中，基于程序频谱的缺陷定位(spectrum-based fault localization,SFL)方法因其具有较好定位效果而被广泛关注。

Reps 等人[19]首次提出程序频谱的概念，后来的研究者将其用于程序分析。程序频谱是结合测试用例对被测程序的覆盖情况和测试执行结果构成的一张二维表，通过对程序频谱的计算，可以获得缺陷的可疑位置。Jones 等人[7,8]首先提出 Tarantula 计算公式，根据程序频谱计算程序组件含有缺陷的可能性。Abreu 等人[9,10]之后提出 Jaccard 计算公式和 Ochiai 计算公式来计算程序组件的可疑分数。文献[14]提出 PRFL 技术，使用 PageRank 算法区分不同测试的贡献程度，从而增强传统 SFL 算法的定位效果。文献[16]针对测试套件中成功用例占比较大问题，提出 EP*方法来调整成功用例数的影响，从而提高缺陷定位精度。

动态缺陷定位方法是以测试用例运行成本为代价的，现有的 SFL 方法直接将缺陷定位至语句，一方面定位时间开销较大，需要收集、分析每个测试用例对每条语句的覆盖情况；另一方面可行性有限，面向对象单元测试常采用的随机测试，很难保证语句的覆盖率，语句的频谱计算有限。本文提出一种基于程序频谱的两阶段缺陷定位方法，基本思路为：第一阶段进行粗粒度定位，选择最流行且具有较高准确率的 Tarantula 算法计算每个程序模块的可疑分数，将缺陷定位到可能的程序模块；第二阶段对于可疑程序模块进行语句细粒度的定位，选择交叉表算法，考虑语句覆盖情况与测试用例执行结果之间的相关关系，计算语句的可疑分数，将缺陷定位到可能的语句，最终生成可疑语句推荐列表，辅助开发人员定位缺陷。实验证明，相比于直接定位至语句的方法，本

收稿日期：2020-03-06；**修回日期：**2020-04-23 **基金项目：**国家自然科学基金资助项目(61772352)

作者简介：伍佳(1996-)，女，四川成都人，硕士，主要研究方向为软件质量保证与测试、软件工程；洪玫(1963-)，女(通信作者)，浙江定海，教授，副院长，硕士，主要研究方向为软件分析与测试(hongmei@scu.edu.cn)；万莹(1993-)，女，贵州安顺人，硕士，主要研究方向为软件质量保证；邓惠心(1996-)，女，四川南充人，硕士，主要研究方向为软件质量保证与测试；潘春霞(1995-)，女，四川内江人，硕士，主要研究方向为软件质量保证与测试.

文方法的定位成本更低。

1　第一阶段——可疑程序模块定位

在面向对象程序中，基本程序模块定义为类的方法。在这一阶段，依据被定位缺陷项目的源代码、测试记录(包括测试覆盖信息及测试用例执行结果)，计算程序模块被测试用例覆盖的频谱，确定可疑的程序模块列表，并按照可疑程度排序。主要包括以下步骤：

a) 基于测试记录，构建程序频谱。

b) 使用 Tarantula 算法计算程序模块的可疑分数。

c) 生成可疑程序模块推荐列表。

1.1　构建程序频谱

文献[20]将程序频谱的构建方式分为：轻量级构建、重量级构建及其他方式三种。本文采用轻量级程序频谱构建方式，即通过简单统计测试用例对软件项目的覆盖信息和执行结果构建程序频谱。

在执行测试用例时，获取测试覆盖信息和执行结果的方法很多，比如可以使用测试覆盖率工具(EclEmma、Cobertura、Jacoco 等)收集并保存为文件形式。解析覆盖率文件并存储每个测试用例对程序模块、语句的覆盖情况，统计每个测试用例对源代码中每个程序模块的覆盖情况，得到程序模块的覆盖矩阵 $T_{m\times n}$，结合测试用例集的执行结果向量 R，生成程序模块粒度的程序频谱 S。

$$S=\begin{bmatrix} T_{m\times n} \\ R \end{bmatrix}=\begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1n} \\ t_{21} & t_{22} & \cdots & t_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ t_{m1} & t_{m2} & \cdots & t_{mn} \\ r_1 & r_2 & \cdots & r_n \end{bmatrix} \tag{1}$$

其中：

a) m 代表源代码中程序模块个数，n 代表测试用例集中测试用例个数。

b) $t_{ij}=1$ 代表程序模块 i 被测试用例 j 覆盖，$t_{ij}=0$ 代表程序模块 i 未被测试用例 j 覆盖。

c) $r_j=1$ 代表测试用例 j 执行失败，$r_j=0$ 代表测试用例 j 执行成功。

1.2　生成可疑程序模块推荐列表

目前，大部分 SFL 研究在设置可疑分数计算公式时都以程序频谱作为基础，并考虑 Wong 等人[12]总结的以下假设：

a) 程序组件的怀疑率与该程序实体被成功测试用例覆盖的次数成反比。

b) 程序组件的怀疑率与该程序实体被失败测试用例覆盖的次数成正比。

c) 程序组件的怀疑率与该程序实体未被失败测试用例覆盖的次数成反比。

d) 在设定怀疑率公式时,应该为假设 b)设置更高的权重。

Jones 等人[7,8]提出 Tarantula 公式计算程序组件中含有缺陷的可能性，计算结果作为该程序组件的可疑分数。他们认为：由大量失败测试用例覆盖的程序组件极有可能含有缺陷，由大量成功测试用例覆盖的程序组件不太可能含有缺陷。给定程序组件 m，计算其可疑分数的公式为

$$\text{Tarantula}(m)=\frac{n_{cf}(m)/n_f}{n_{cf}(m)/n_f+n_{cs}(m)/n_s} \tag{2}$$

其中：

a) $n_{cf}(m)$ 和 $n_{cs}(m)$ 表示覆盖程序组件 m 的失败测试用例数量和成功测试用例数量。

b) n_f 和 n_s 表示测试用例集中失败测试用例数量和成功测试用例数量。

Tarantula(m) 的取值范围为[0,1]，分数越高代表该程序组件含有缺陷的可能性越大。本文以 1.1 节构建的程序频谱 S 作为 Tarantula 算法的输入，计算源代码中每个程序模块的可疑分数，按照分数由高到低排列，生成可疑程序模块推荐列表 M，包含排名、程序模块名(“类名->方法名->参数名”)以及可疑分数，图 1 为列表 M 实例。

2　第二阶段——可疑语句定位

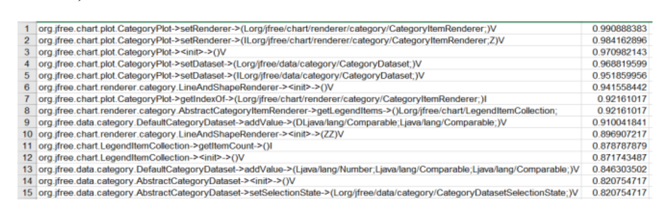
第一阶段输出可疑程序模块推荐列表 M，若以此作为缺陷定位参考，开发人员仍会有较大的代码阅读量。因此，需要进一步分析模块中哪些语句可能含有缺陷。可疑语句定位阶段以前一阶段保存的语句覆盖情况、执行结果和可疑程序模块推荐列表 M 作为输入，输出可疑语句推荐列表 T。包括以下步骤：

a) 计算可疑分数阈值。

b) 基于语句覆盖情况和执行结果，构建交叉表。

c) 使用交叉表算法计算语句的可疑分数。

d) 生成可疑语句推荐列表。



1	org.jfree.chart.plot.CategoryPlot->setRenderer->(Lorg/jfree/chart/renderer/category/CategoryItemRenderer;)V	0.990888383
2	org.jfree.chart.plot.CategoryPlot->setRenderer->(ILorg/jfree/chart/renderer/category/CategoryItemRenderer;Z)V	0.984162896
3	org.jfree.chart.plot.CategoryPlot-><init>->()V	0.970982143
4	org.jfree.chart.plot.CategoryPlot->setDataset->(Lorg/jfree/data/category/CategoryDataset;)V	0.968819599
5	org.jfree.chart.plot.CategoryPlot->setDataset->(ILorg/jfree/data/category/CategoryDataset;)V	0.951859956
6	org.jfree.chart.renderer.category.LineAndShapeRenderer-><init>->()V	0.941558442
7	org.jfree.chart.plot.CategoryPlot->getIndexOf->(Lorg/jfree/chart/renderer/category/CategoryItemRenderer;)I	0.92161017
8	org.jfree.chart.renderer.category.AbstractCategoryItemRenderer->getLegendItems->()Lorg/jfree/chart/LegendItemCollection;	0.92161017
9	org.jfree.data.category.DefaultCategoryDataset->addValue->(DLjava/lang/Comparable;Ljava/lang/Comparable;)V	0.910041841
10	org.jfree.chart.renderer.category.LineAndShapeRenderer-><init>->(ZZ)V	0.896907217
11	org.jfree.chart.LegendItemCollection->getItemCount->()I	0.878787879
12	org.jfree.chart.LegendItemCollection-><init>->()V	0.871743487
13	org.jfree.data.category.DefaultCategoryDataset->addValue->(Ljava/lang/Number;Ljava/lang/Comparable;Ljava/lang/Comparable;)V	0.846303502
14	org.jfree.data.category.AbstractCategoryDataset-><init>->()V	0.820754717
15	org.jfree.data.category.AbstractCategoryDataset->setSelectionState->(Lorg/jfree/data/category/CategoryDatasetSelectionState;)V	0.820754717

图 1　可疑程序模块推荐列表 M 实例

Fig. 1　Example of suspicious program module recommendation list M

2.1　计算可疑分数阈值

当程序模块数量较多时，若要分析每个模块中的每条语句将会带来巨大的计算开销。因此，需要舍弃不太可能含有缺陷的模块，本文利用可疑分数阈值过滤掉那些不太可能含有缺陷的程序模块，对剩余模块再进行语句粒度的定位分析。

文献[21]中将可疑分数阈值定为 $\sigma=0.7$，即对于任何软件项目，可疑分数大于等于 0.7 的程序模块才进一步分析。本文使用 $\sigma=0.7$ 在 Defects4j[22]中的 355 个真实缺陷上进行实验，统计结果显示有 22.54%的真实缺陷所在程序模块的可疑分数小于 0.7。因此，阈值选择不当，会影响对部分缺陷的定位。

为了更有效确定阈值，本文提出一种自动计算可疑分数阈值的方法。在第一阶段输出的列表 M 中，丢弃可疑分数为 0 的模块，保留可疑分数为 1 的模块。对于其他模块，本文认为只有可疑分数大于等于平均值的才值得进一步分析。因此，本文使用以下公式计算可疑分数阈值：

$$\sigma=\frac{1}{k}\times\sum_{i=1}^{k}\text{Tarantula}(m_i) \tag{3}$$

其中：k 为列表 M 中，可疑分数不为 0 和 1 的程序模块个数。经实验验证，使用平均值作为阈值时，只有 1.97%的缺陷因阈值过大导致定位失败，相比文献[21]中使用的 $\sigma=0.7$ 降低了 20.57%。

本文使用式(3)计算每个缺陷项目的可疑分数阈值，根据计算结果对第一阶段产生的可疑模块推荐列表 M 进行筛选，保留可疑分数大于等于阈值的程序模块，得到程序模块集 M1。

2.2　构建交叉表

交叉表是统计学中常用的一种分类汇总表，交叉表分析通常用于研究两个或多个分类变量之间的关系。Wong 等人[11]将交叉表分析应用到缺陷定位研究中，为每条语句构建交叉表，计算语句含有缺陷的可能性。

统计每个测试用例对 M1 中每条语句的覆盖情况及执行结果，形成语句粒度的覆盖矩阵 $C_{s\times n}$。结合执行结果向量 R，构建语句粒度的程序频谱，为每条语句生成交叉表 Q，含有列分类变量“测试用例覆盖语句情况”和行分类变量“测试

用例运行结果”。

如表 1 所示，其中 $N_{CS(s_i)}$ 代表覆盖语句 s_i 并运行成功的测试用例个数。

表 1　语句 s_i 的交叉表 Q

Tab. 1　Crosstab Q of program statement s_i

运行结果	语句覆盖情况		
	覆盖语句 s_i	未覆盖语句 s_i	合计
运行成功	$N_{CS(s_i)}$	$N_{US(s_i)}$	N_S
运行失败	$N_{CF(s_i)}$	$N_{UF(s_i)}$	N_F
合计	$N_{C(s_i)}$	$N_{U(s_i)}$	N

2.3　生成可疑语句推荐列表

在交叉表算法[11]中，输入每条语句的交叉表，通过以下步骤计算每条语句含有缺陷的可能性。

a) 通过假设检验考察分类变量之间的相关情况，使用以下公式计算卡方值 χ^2， χ^2 越大代表分类变量越相关。

$$\chi^2(s_i)=\frac{(N_{CF(s_i)}-E_{CF(s_i)})^2}{E_{CF(s_i)}}+\frac{(N_{CS(s_i)}-E_{CS(s_i)})^2}{E_{CS(s_i)}}+\frac{(N_{UF(s_i)}-E_{UF(s_i)})^2}{E_{UF(s_i)}}+\frac{(N_{US(s_i)}-E_{US(s_i)})^2}{E_{US(s_i)}} \tag{4}$$

其中：

$$\begin{cases} E_{CF(s_i)}=\dfrac{N_{C(s_i)}/N_F}{N} \\ E_{CS(s_i)}=\dfrac{N_{C(s_i)}/N_S}{N} \\ E_{UF(s_i)}=\dfrac{N_{U(s_i)}/N_F}{N} \\ E_{US(s_i)}=\dfrac{N_{U(s_i)}/N_S}{N} \end{cases} \tag{5}$$

b) 使用式(6)计算分类变量之间的相关关系。

$$\phi(s_i)=\frac{N_{CF(s_i)}/N_F}{N_{CS(s_i)}/N_S} \tag{6}$$

如果 $\phi(s_i)=1$，代表语句 s_i 的覆盖与执行结果之间完全独立，可以认为语句 s_i 的覆盖对成功执行和失败执行有着相同的贡献值；如果 $\phi(s_i)>1$，代表语句 s_i 的覆盖与失败执行更相关；反之代表代表语句 s_i 的覆盖与成功执行更相关。

c) 结合式(4)和(6)的计算结果，计算每条语句的可疑分数。

$$\text{crosstab}(s_i)=\begin{cases} \chi^2(s_i) & \phi(s_i)>1 \\ 0 & \phi(s_i)=1 \\ -\chi^2(s_i) & \phi(s_i)<1 \end{cases} \tag{7}$$

根据 2.2 节构建的语句交叉表，计算每条语句的 *crosstab*(*s*) 值作为可疑分数，分数越高的语句越有可能为真实缺陷语句。并从高到低排序，生成可疑语句推荐列表 T(排名、程序模块名、语句行号以及可疑分数)，用于辅助开发人员进行缺陷定位。图 2 为可疑语句推荐列表 T 实例。

1	org.jfree.chart.renderer.category.AbstractCategoryItemRenderer->getLegendItems->()Lorg/jfree/chart/LegendItemCollection;	1793	436
2	org.jfree.chart.plot.CategoryPlot->setRenderer->(Lorg/jfree/chart/renderer/category/CategoryItemRenderer;)V	1613	86.39816092
3	org.jfree.chart.plot.CategoryPlot->setRenderer->(Lorg/jfree/chart/renderer/category/CategoryItemRenderer;)V	1614	86.39816092
4	org.jfree.chart.plot.CategoryPlot->setRenderer->(ILorg/jfree/chart/renderer/category/CategoryItemRenderer;Z)V	1665	53.62298851
5	org.jfree.chart.plot.CategoryPlot->setRenderer->(ILorg/jfree/chart/renderer/category/CategoryItemRenderer;Z)V	1667	53.62298851
6	org.jfree.chart.plot.CategoryPlot->setRenderer->(ILorg/jfree/chart/renderer/category/CategoryItemRenderer;Z)V	1672	53.62298851
7	org.jfree.chart.plot.CategoryPlot-><init>->()V	567	30.21215107
8	org.jfree.chart.plot.CategoryPlot-><init>->()V	568	30.21215107
9	org.jfree.chart.plot.CategoryPlot->setDataset->(Lorg/jfree/data/category/CategoryDataset;)V	1339	28.13118774
10	org.jfree.chart.plot.CategoryPlot->setDataset->(Lorg/jfree/data/category/CategoryDataset;)V	1340	28.13118774
11	org.jfree.chart.plot.CategoryPlot->setDataset->(ILorg/jfree/data/category/CategoryDataset;)V	1358	18.86144201
12	org.jfree.chart.plot.CategoryPlot->setDataset->(ILorg/jfree/data/category/CategoryDataset;)V	1367	17.9978011
13	org.jfree.chart.renderer.category.LineAndShapeRenderer-><init>->()V	201	14.60492611
14	org.jfree.chart.renderer.category.LineAndShapeRenderer-><init>->()V	202	14.60492611
15	org.jfree.chart.plot.CategoryPlot->getIndexOf->(Lorg/jfree/chart/renderer/category/CategoryItemRenderer;)I	1727	10.49776165

图 2　可疑语句推荐列表 T 实例

Fig. 2　Example of suspicious statement recommendation list T

3　实验验证

本文通过分析总结现有文献，选取了 GitHub 平台中的真实缺陷库 Defects4j 作为实验数据集。Defects4j 是由 Just 等人[22]提出的一个可扩展框架，该框架包含了 5 个 java 开源项目的多个真实缺陷，并将每个项目的单个缺陷看做一个缺陷版本。Defects4j 为每个真实缺陷都提供了一组开发人员编写的测试用例集，该测试用例集中至少包含一个执行失败的测试用例。本文方案要求在测试用例集中至少有一个失败的测试用例和一个成功的测试用例，因此舍弃了 2 个不适用于本文方案的缺陷版本，分别为 Chart-10b、Lang-57b，它们的测试用例集中都没有成功的测试用例。剩余 355 个缺陷版本具体信息如表 2 所示。

表 2　实验项目信息

Tab. 2　Information of experimental projects

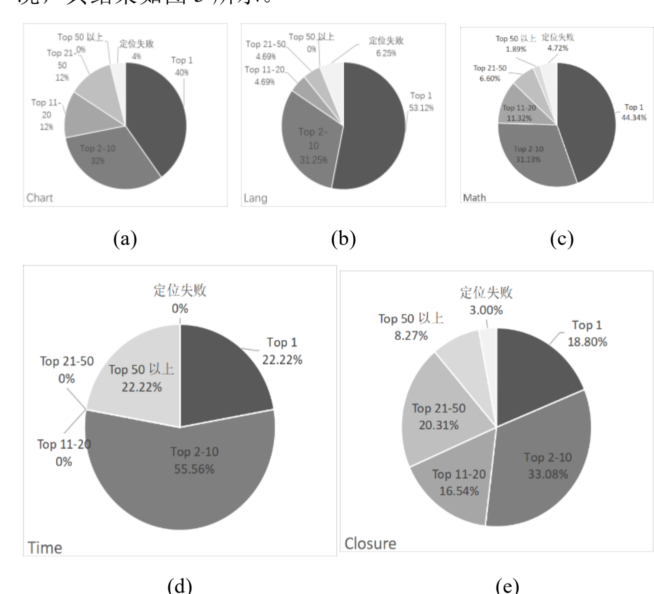
项目名	缺陷版本个数	平均代码行数	平均测试用例个数	平均失败测试用例个数
Chart	25	4187	230	4
Lang	64	814	176	2
Math	106	2266	215	2
Time	27	5218	2606	3
Closure	133	16483	1540	3

3.1　方案可行性实验验证

实验设计：Defects4j 中的 355 个缺陷版本分别执行其对应的测试用例集并使用 Cobertura 工具收集每个测试用例的覆盖信息和执行结果。利用本文提出的方案分别进行缺陷定位，并对结果进行评价。

评价指标：本实验使用 Top N 作为评价指标，表示对一个项目的多个缺陷版本进行定位后，真实缺陷语句在推荐列表中的排名为前 N 的情况次数。

实验结果及分析：本实验对 Defects4j 中的 5 个项目缺陷版本进行缺陷定位，结合 Defects4j 提供的真实缺陷位置信息，使用 top N 作为评价指标，划分了 6 个段，分别是真实缺陷在可疑语句列表的第 1 位、第 2-10 位、第 11-20 位、第 21-50 位、第 50 位以上，分别针对 5 个项目统计了缺陷定位情况，其结果如图 3 所示。



(a)　(b)　(c)

(d)　(e)

图 3　Defects4j 中各项目定位结果分布

Fig. 3　Distributions of project fault localization results in Defects4j

在图 3 中(a)至(e)分别表示每个项目的定位结果中，真实缺陷在各个分段的分布情况。经统计，有 68%的缺陷在推荐列表的前 10 中列出，有 34%的缺陷在推荐列表的首位定位。证明本文方案具有一定可行性。

对于定位失败的 14 个项目版本，分析发现了两个原因：

a) 项目的缺陷为在源代码中丢失了一个完整的程序模块，导致在第一阶段产生的可疑程序模块列表中没有该模块的信息。

b) 项目的测试用例集中有大量不稳定(偶然正确)测试用例，即真实缺陷所在程序模块被大量成功测试用例覆盖，可疑分数小于阈值，导致在第二阶段中丢失了真实缺陷所在的模块。

3.2　方案有效性实验验证

实验设计：比较实验，使用目前流行的缺陷定位算法 Jasscard[9,10]、Ochicai[9,10]、Dstar2[12]、Op2[13]、EP3[20]对 355 个真实缺陷进行定位，并将每个算法的定位结果与本文方案进行对比分析。

评价指标：采用定位时间和定位支出这两个指标对实验结果进行评估。

a) 定位时间：在输入缺陷项目源代码及测试记录时、输出可疑语句推荐列表时分别获取当前系统时间，两者之差作为定位时间。

b) 定位支出(Expense)：根据定位结果列表，开发人员找到真正缺陷所需的支出，其计算公式为

$$\text{Expense} = n \times \frac{1}{N} \tag{8}$$

其中：n 表示真正缺陷语句在结果列表中的排名，若多个语句拥有相同的可疑分数，本文取他们的平均排名。N 表示程序总语句数。

实验结果及分析：本实验统计了每种方法的定位时间，并将本文方案分别与 Jasscard、Ochicai、Dstar^2、Op2、EP^3 五种算法进行对比，其结果如表 3 所示。

表 3　定位时间对比

Tab. 3　Comparison of fault localization time

项目名称(平均代码行/平均测试用例数)	本文方案与其他算法相比，定位时间增加/减少				
	与 Jasscard 相比	与 Ochiai 相比	与 Dstar^2 相比	与 Op2 相比	与 EP^3 相比
Chart (4187/230)	↓ 10.44%	↓ 10.78%	↓ 10.30%	↓ 11.15%	↓ 10.65%
Lang (814/176)	↓ 7.25%	↓ 5.83%	↓ 8.30%	↓ 9.65%	↓ 8.37%
Math (2266/215)	↓ 9.22%	↓ 8.42%	↓ 9.99%	↓ 10.74%	↓ 10.35%
Time (5218/2606)	↓ 10.61%	↓ 10.97%	↓ 10.77%	↓ 11.36%	↓ 11.22%
Closure (16483/1540)	↓ 11.75%	↓ 11.74%	↓ 11.93%	↓ 11.84%	↓ 12.25%

由表 3 可知，本文方案与其他定位算法相比，定位时间最低减少 5.83%，最高减少 12.25%，平均减少 10.24%。并且，随着软件项目规模的增大，定位的时间花销减少的比例越大。因此，本文方案能够有效地减少缺陷定位的时间花销，并在大型复杂软件中表现更为突出。

为了更好地说明本文方案的整体有效性，本实验统计了每种定位方法的定位支出，其结果如图 4 所示。横轴表示 Defects4j 中的 5 个项目，纵轴表示各方法的平均定位支出。对于 355 个缺陷版本，Jasscard、Ochiai、Dstar^2、Op2、EP^3 的平均定位支出分别为 0.0431、0.0405、0.0404、0.0476、0.0402，本文方案的平均定位支出为 0.0417，比 Jasscard、Op2 减少了 3.25%、12.39%，比 Ochiai、Dstar^2、EP^3 增加了 2.96%、3.22%、3.73%。

结合表 3 和图 4 的分析结果，能够证明本文方案能够在保证定位效果的同时，有效地减少定位的时间花销。

4　结束语

本文在现有的 SFL 方法基础上加以改进，针对面向对象软件提出一种“两阶段”的自动化缺陷定位方法。通过与现有流行的 SFL 方法进行对比，本文方法能够在保证缺陷定位效果的同时减少缺陷定位的成本，节省了开发人员的调试时间。针对本文方法还存在一些问题需要进一步研究与改进：

a) 本文方法主要针对单缺陷项目，但在实际项目中缺陷的存在往往是多个且有联系的，未来可以进一步的研究多缺陷定位问题。

b) 本文采用轻量级程序频谱构造方法，忽略了程序组件间的控制及数据依赖关系，在未来研究中可考虑采用重量级程序频谱构造方法，从而提高定位效果。

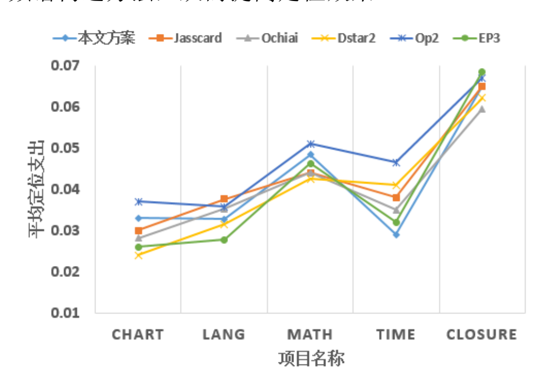


图 4　定位支出对比

Fig. 4　Comparison of fault localization expense

参考文献：

[1] Youm K C, Ahn J, Lee E. Improved bug localization based on code change histories and bug reports [J]. Information and Software Technology, 2016, 82: 177-192.

[2] Amar A, Rigby P C. Mining historical test logs to predict bugs and localize faults in the test logs [C]// Proc of the 41st International Conference on Software Engineering. Piscataway, NJ: IEEE Press, 2019: 140-151.

[3] Li Xia, Li Wei, Zhang Yuqun, *et al.* DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization [C]// Proc of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM Press, 2019: 169-180.

[4] Loyola P, Gajananan K, Satoh F. Bug Localization by Learning to Rank and Represent Bug Inducing Changes [C]// Proc of the 27th ACM International Conference on Information and Knowledge Management. New York: ACM Press, 2018: 657-665.

[5] Wotawa F. Fault Localization Based on Dynamic Slicing and Hitting-Set Computation [C]// Proc of the 10th International Conference on Quality Software. Washington DC: IEEE Computer Science, 2010: 161-170.

[6] Surendran A, Samuel P. Fault localization using forward slicing spectrum [C]// Proc of Research in Adaptive and Convergent Systems. New York: ACM Press, 2013: 397-398.

[7] Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization [C]// Proc of the 24th International Conference on Software Engineering. New York: ACM Press, 2002: 467-477.

[8] Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique [C]// Proc of the 20th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM Press, 2005: 273-282.

[9] Abreu R, Zoeteweij P, Van Gemund A J C. An evaluation of similarity coefficients for software fault localization [C]// Proc of the 12th Pacific Rim International Symposium on Dependable Computing. Washington DC: IEEE Computer Science, 2006: 39-46.

[10] Abreu R, Zoeteweij P, Van Gemund A J C. On the accuracy of spectrum-based fault localization [C]// Proc of the Testing: Academic and Industrial Conference Practice and Research Techniques. Washington DC: IEEE Computer Science, 2007: 89-98.

[11] Wong W E, Debroy V, Xu D. Towards better bug localization: a crosstab-based statistical approach [J]. IEEE Trans on Systems Man and

Cybernetics Part C, 2012, 42 (3): 378-396.

[12] Wong W E, Debroy V, Gao R, *et al.* The DStar method for effective software fault localization [J]. IEEE Trans on Reliability, 2014, 63 (1): 290–308.

[13] Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis [J]. ACM Trans on Software Engineering and Methodology, 2011, 20 (3): 1-32.

[14] Zhang Mengshi, Li Xia, Zhang Lingming, *et al.* Boosting spectrum-based fault localization using PageRank [C]// Proc of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM Press, 2017: 261-272.

[15] Kim J, Kim J, Lee E. A novel variable-centric fault localization technique [C]// Proc of the 40th International Conference on Software Engineering: Companion Proceeedings. New York: ACM Press, 2018: 252-253.

[16] 蔡蕊，张仕，余晓菲，等. 基于程序频谱的缺陷定位方法 [J]. 计算机系统应用, 2019, 28 (1): 188－193. (Cai Rui, Zhang Shi, Yu Xiaofei, *et al.* Defect Localization Method Based on Program Spectrum [J]. Computer Systems and Applications, 2019, 28 (1): 188－193.)

[17] Le T B, Oentaryo R J, Lo D. Information retrieval and spectrum based bug localization: better together [C]// Proc of the 10th Joint Meeting on Foundations of Software Engineering. New York: ACM Press, 2015: 579-590.

[18] Lam A N, Nguyen A T, Nguyen H A, *et al.* Bug Localization with Combination of Deep Learning and Information Retrieval [C]// Proc of the 25th International Conference on Program Comprehension. Piscataway, NJ: IEEE Press, 2017: 218-229.

[19] Reps T, Ball T, Das M, *et al.* The use of program profiling for software maintenance with applications to the year 2000 problem [J]. ACM SIGSOFT Software Engineering Notes, 1997, 22 (6): 432-449.

[20] 陈翔，鞠小林，文万志，等. 基于程序频谱的动态缺陷定位方法研究 [J]. 软件学报, 2015 (2): 390-412. (Chen Xiang, Ju Xiaolin, Wen Wanzhi, *et al.* Review of dynamic fault localization approaches based on program spectrum [J]. Journal of Software, 2015 (2): 390-412.)

[21] Perez A, Abreu R, Riboira A. A dynamic code coverage approach to maximize fault localization efficiency [J]. Journal of Systems and Software, 2014, 90: 18-28.

[22] Just R, Jalali D, Ernst M D. Defects4J: a database of existing faults to enable controlled testing studies for Java programs [C]// Proc of International Symposium on Software Testing and Analysis. New York: ACM Press, 2014: 437-440.